Linux debugging, profiling, tracing and performance analysis training

Course duration

4 half days – 16 hours

Language

Materials	English
Oral Lecture	English
	French
	Italian

Trainer

One of the following engineers

- Alexis Lothoré
- Luca Ceresoli

Contact

training@bootlin.com
+33 484 258 097

bootlin
bootlin.com

Online seminar

Audience

Companies and engineers interested in debugging, profiling and tracing Linux systems and applications, to analyze and address performance or latency problems.

Training objectives

- Be able to understand the main concepts of Linux that are relevant for performance analysis: process, threads, memory management, virtual memory, execution contexts, etc.
- Be able to analyze why a system is loaded and what are the elements that contributes to this load using common Linux observability tools.
- Be able to debug an userspace application using *gdb*, either live or after a crash, and analyze the contents of ELF binaries.
- Be able to trace and profile a complete userspace application and its interactions with the Linux kernel in order to fix bugs using *strace*, *ltrace*, *perf* or *Callgrind*.
- Be able to understand classical memory issues and analyze them using *valgrind*, *libefence* or *Massif*.
- Be able to trace and profile the entire Linux system, using *perf*, *ftrace*, *kprobes*, *eBPF* tools, *kernelshark* or *LTTng*
- Be able to debug Linux kernel issues: debug kernel crashes live or post-mortem, analyze memory issues at the kernel level, analyze locking issues, use kernel-level debuggers.

Prerequisites

- **Knowledge and practice of UNIX or GNU/Linux commands**: participants must be familiar with the Linux command line. Participants lacking experience on this topic should get trained by themselves, for example with our freely available on-line slides.
- **Minimal experience in embedded Linux development**: participants should have a minimal understanding of the architecture of embedded Linux systems: role of the Linux kernel vs. user-space, development of Linux user-space applications in C. Following Bootlin's Embedded Linux course allows to fulfill this pre-requisite.
- **Minimal English language level: B1**, according to the *Common European Framework of References for Languages*, for our sessions in English. See the CEFR grid for self-evaluation.

Pedagogics

- Lectures delivered by the trainer, over video-conference. Participants can ask questions at any time.
- Practical demonstrations done by the trainer, based on practical labs, over video-conference. Participants can ask questions at any time. Optionally, participants who have access to the hardware accessories can reproduce the practical labs by themselves.
- Instant messaging for questions between sessions (replies under 24h, outside of week-ends and bank holidays).
- Electronic copies of presentations, lab instructions and data files. They are freely available here.

Certificate

Only the participants who have attended all training sessions, and who have scored over 50% of correct answers at the final evaluation will receive a training certificate from Bootlin.

Disabilities

Participants with disabilities who have special needs are invited to contact us at *training@bootlin.com* to discuss adaptations to the training course.

Required equipement

Mandatory equipment:

- Computer with the operating system of your choice, with the Google Chrome or Chromium browser for videoconferencing.
- Webcam and microphone (preferably from an audio headset).
- High speed access to the Internet.

Optionnally, if the participants want to be able to reproduce the practical labs by themselves, they must separately purchase the hardware platform and accessories, and must have a PC computer with a native installation of Ubuntu Linux 24.04.

Hardware platform for practical labs

STM32MP1 Discovery Kit

One of these Discovery Kits from STMicroelectronics: **STM32MP157A-DK1**, **STM32MP157D-DK1**, **STM32MP157C-DK2** or **STM32MP157F-DK2**

- STM32MP157, dual Cortex-A7 processor from STMicroelectronics
- USB powered
- 512 MB DDR3L RAM
- Gigabit Ethernet port
- 4 USB 2.0 host ports
- 1 USB-C OTG port
- 1 Micro SD slot
- On-board ST-LINK/V2-1 debugger
- Arduino compatible headers
- Audio codec, buttons, LEDs
- LCD touchscreen (DK2 kits only)

Training Schedule

Half day 1

Lecture	Linux application stack	▪ Global picture: understanding the general architecture of a Linux system, overview of the major components. ▪ What is the difference between a process and a thread, how applications run concurrently. ▪ ELF files and associated analysis tools. ▪ Userspace application memory layout (heap, stack, shared libraries mappings, etc). ▪ MMU and memory management: physical/virtual address spaces. ▪ Kernel context switching and scheduling ▪ Kernel execution contexts: kernel threads, workqueues, interrupt, threaded interrupts, softirq
Lecture	Common analysis & observability tools	▪ Analyzing an ELF file with GNU binary utilities (*objdump*, *addr2line*). ▪ Tools to monitor a Linux system: processes, memory usage and mapping, resources. ▪ Using *vmstat*, *iostat*, *ps*, *top*, *iotop*, *free* and understanding the metrics they provide. ▪ Pseudo filesystems: *procfs*, *sysfs* and *debugfs*.
Demo	Check what is running on a system and its load	▪ Observe running processes using *ps* and *top*. ▪ Check memory allocation and mapping with *procfs* and *pmap*. ▪ Monitor other resources usage using *iostat*, *vmstat* and *netstat*.
Lecture	Debugging an application	▪ Using *gdb* on a live process. ▪ Understanding compiler optimizations impact on debuggability. ▪ Postmortem diagnostic using core files. ▪ Remote debugging with *gdbserver*. ▪ Extending *gdb* capabilities using python scripting

Half day 2

Demo	Solving an application crash	▪ Analysis of compiled C code with compiler-explorer to understand optimizations. ▪ Managing *gdb* from the command line, then from an IDE. ▪ Using *gdb* Python scripting capabilities. ▪ Debugging a crashed application using a coredump with *gdb*.
Lecture	Tracing an application	▪ Tracing system calls with *strace*. ▪ Tracing library calls with *ltrace*. ▪ Overloading library functions using *LD_PRELOAD*.
Demo	Debugging application issues	▪ Analyze dynamic library calls from an application using *ltrace*. ▪ Overloading library functions using *LD_PRELOAD*. ▪ Analyzing an application system calls using *strace*.
Lecture	Memory issues	▪ Usual memory issues: buffer overflow, segmentation fault, memory leaks, heap-stack collision. ▪ Memory corruption tooling, *valgrind*, *libefence*, etc. ▪ Heap profiling using *Massif* and *heaptrack*
Demo	Debugging memory issues	▪ Memory leak and misbehavior detection with *valgrind* and *vgdb*. ▪ Visualizing application heap using *Massif*.

Half day 3

Lecture	Application profiling	▪ Performances issues. ▪ Gathering profiling data with *perf*. ▪ Analyzing an application callgraph using *Callgrind* and *KCachegrind*. ▪ Interpreting the data recorded by *perf*.
Demo	Application profiling	▪ Profiling an application with *Callgrind*/*KCachegrind*. ▪ Analyzing application performance with *perf*. ▪ Generating a flamegraph using *FlameGraph*.
Lecture	System wide profiling and tracing	▪ System wide profiling using *perf*. ▪ Using *kprobes* to hook on kernel code without recompiling. ▪ Application and kernel tracing and visualization using *ftrace*, *kernelshark* or *LTTng* ▪ Tracing with *eBPF*: core principles, usage with BCC and with libbpf
Demo	System wide profiling and tracing	▪ System profiling with *perf*. ▪ System wide latencies debugging using *ftrace* and *kernelshark*.
Half day 4		
Demo	Tracing tool with eBPF	▪ Python scripting with *bcc*. ▪ Custom tool development with libbpf.
Lecture	Kernel debugging	▪ Kernel compilation results (`vmlinux`, `System.map`). ▪ Understanding and configuring kernel *oops* behavior. ▪ Post mortem analysis using kernel crash dump with *crash*. ▪ Memory issues (*KASAN*, *UBSAN*, *Kmemleak*). ▪ Debugging the kernel using *KGDB* and *KDB*. ▪ Kernel locking debug configuration options (lockdep). ▪ Other kernel configuration options that are useful for debug.
Demo	Kernel debugging	▪ Analyzing an *oops* after using a faulty module with *obdjump* and *addr2line*. ▪ Debugging a deadlock problem using *PROVE_LOCKING* options. ▪ Detecting undefined behavior with *UBSAN* in kernel code. ▪ Find a module memory leak using *kmemleak*. ▪ Debugging a module with *KGDB*.